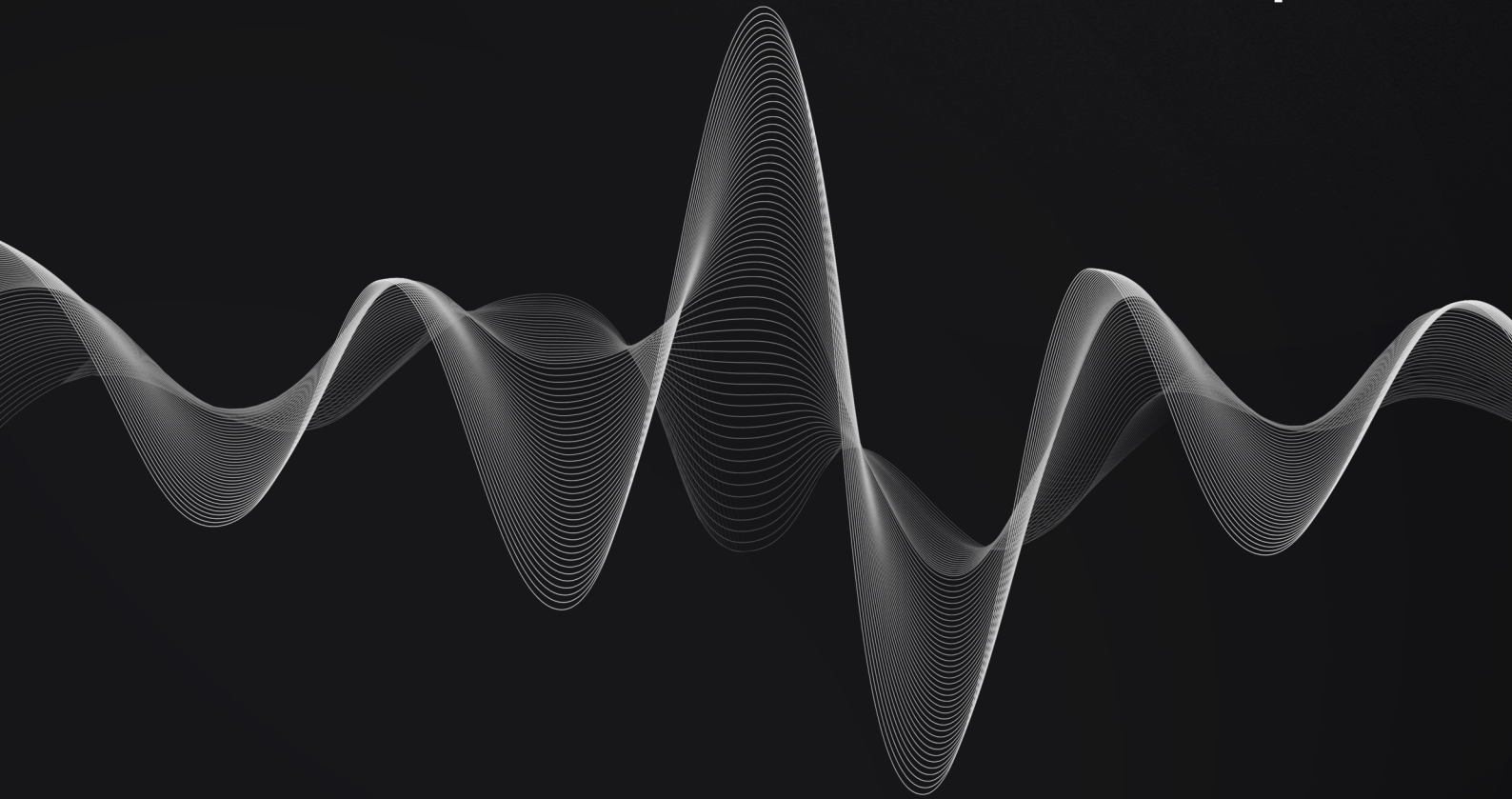


BonusBlock

Smart Contracts Audit Report









Document Control

PUBLIC

FINAL_(v2.0)

Audit_Report_BBLK-SCA_FINAL_20

Apr 1, 2024		v0.1	Michal Bajor: Initial draft
Apr 2, 2024		v0.2	Michal Bajor: Added findings
Apr 2, 2024		v0.3	João Simões: Added findings
Apr 2, 2024		v1.0	Charles Dray: Approved
Apr 5, 2024		v1.1	João Simões: Reviewed findings
Apr 11, 2024		v2.0	Charles Dray: Published

Points of Contact

Oskars Jepsis
Luke Forzley
Charles Dray

BonusBlock
Trivium
Resonance

oskars@bonusblock.io
luciusforzley@gmail.com
charles@resonance.security

Testing Team

Michal Bajor
João Simões
Ilan Abitbol
Michał Bazyli

Resonance
Resonance
Resonance
Resonance

michal.bajor@resonance.security
joao.simoes@resonance.security
ilan.abitbol@resonance.security
michal.bazyli@resonance.security

Copyright and Disclaimer

© 2024 Resonance Security, Inc. All rights reserved.

The information in this report is considered confidential and proprietary by Resonance and is licensed to the recipient solely under the terms of the project statement of work. Reproduction or distribution, in whole or in part, is strictly prohibited without the express written permission of Resonance.

All activities performed by Resonance in connection with this project were carried out in accordance with the project statement of work and agreed-upon project plan. It's important to note that security assessments are time-limited and may depend on information provided by the client, its affiliates, or partners. As such, the findings documented in this report should not be considered a comprehensive list of all security issues, flaws, or defects in the target system or codebase.

Furthermore, it is hereby assumed that all of the risks in electing not to remedy the security issues identified henceforth are sole responsibility of the respective client. The acknowledgement and understanding of the risks which may arise due to failure to remedy the described security issues, waives and releases any claims against Resonance, now known or hereafter known, on account of damage or financial loss.

Contents

1 Document Control	2
Copyright and Disclaimer	2
2 Executive Summary	4
System Overview	4
Repository Coverage and Quality.....	4
3 Target	5
4 Methodology	6
Severity Rating.....	7
Repository Coverage and Quality Rating.....	8
5 Findings	9
Campaign Takeover With Possible Griefing	10
Possibility Of Locking Tokens In Inactive Campaigns	11
Incorrect Usage Of Nonce In Signature Replay Verification	12
Possibility of Funding Non-Existent Campaigns	13
Missing Public Key Validation	14
Unused Functions.....	15
Floating Pragma	16
Missing Usage Of ERC165 Standard.....	17
Redundant Code On verifySignature()	18
Possible Reentrancy In safeMint()	19
Unclaimable Ether Sent To Contract.....	20
A Proof of Concepts	21

Executive Summary

Bonusblock contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between March 27, 2024 and April 3, 2024. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated 2 engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted 5 days to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide Bonusblock with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.



System Overview

The audited system comprises a set of repositories containing multiple smart contracts, that implement various staking capabilities for BonusBlock. Two of the repositories are more relevant as they implement, firstly, a vault with the possibility of registering campaigns, staking, funding, and distributing tokens, and secondly, a staking contract with locking and vesting of rewards in block time slots.

There are also other contracts that implement a custom ERC20 token with fixed supply, an ERC721 token mintable with signature verification capabilities, a spotlight that keeps track of payments to beneficiaries, and an improved staking subscription with canceling capabilities.

Off-chain components further include public key and signature generation utilities that will be used and verified during on-chain transactions.



Repository Coverage and Quality

This section of the report has been redacted by the request of the customer.

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: [TriviumNode/bonus-block-rewards-evm/contracts](#)
- Hash: 0b4144f69908781cbf57c170e42c3b9a352a4f13
- Repository: [TriviumNode/bonusblock-spotlight](#)
- Hash: cafdc7821ef456c319ec12c101937c4a2c40f409
- Repository: [TriviumNode/bonus-block721/contracts](#)
- Hash: 67d6408d133155a82d7f0b4c297413a3ef1691f1
- Repository: [TriviumNode/bonusblock-staking](#)
- Hash: 361eba5ec99b078225a8120aac5ac108d591cb63
- Repository: [TriviumNode/custom-destination-erc20](#)
- Hash: 73c1a0409a136f7e1a9a52a8e7bcb9ec32b983a8
- Repository: [TriviumNode/bonusblock-subscription](#)
- Hash: 173e4db49ffe2f066a86ed361516fe1e07a11fea

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process
- Financial related attacks

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks
- Frontrunning attacks
- Unsafe external calls
- Unsafe third party integrations
- Denial of service
- Access control issues

- Inaccurate business logic implementations
- Incorrect gas usage
- Arithmetic issues
- Unsafe callbacks
- Timestamp dependence
- Mishandled panics, errors and exceptions



Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss
2. Medium - Temporary or partial damage or loss
3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions
2. Likely - Requires technical knowledge or no special conditions
3. Very Likely - Requires trivial knowledge or effort or no conditions

		Likelihood		
		Very Likely	Likely	Unlikely
Impact	Strong	Critical	High	Medium
	Medium	High	Medium	Low
	Weak	Medium	Low	Info



Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- ||||| **"Quick Win"** Requires little work for a high impact on risk reduction.
- |||| **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- ||| **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

Findings ID	Description	Severity	Status
RES-01	Campaign Takeover With Possible Griefing		Resolved
RES-02	Possibility Of Locking Tokens In Inactive Campaigns		Resolved
RES-03	Incorrect Usage Of Nonce In Signature Replay Verification		Resolved
RES-04	Possibility of Funding Non-Existent Campaigns		Resolved
RES-05	Missing Public Key Validation		Resolved
RES-06	Unused Functions		Resolved
RES-07	Floating Pragma		Resolved
RES-08	Missing Usage Of ERC165 Standard		Acknowledged
RES-09	Redundant Code On verifySignature()		Acknowledged
RES-10	Possible Reentrancy In safeMint()		Acknowledged
RES-11	Unclaimable Ether Sent To Contract		Resolved



Campaign Takeover With Possible Griefing

Critical

RES-BBLK-SCA01

Business Logic

Resolved

Code Section

- `contracts/MyToken.sol#L93`

Description

The RewardsBank contract allows users to register campaigns. Each campaign requires a user to lock funds in exchange for reward tokens. Any user can register a new campaign, however only the contract's owner can fund the campaign, i.e. increase the rewards. The `registerCampaign` function, responsible for creating new campaigns, contains a validation logic that was meant to prevent recreating an already existing campaign. However, it was identified that the `require` statement implementing this logic will always return `true`, making the whole validation ineffective. As a consequence, multiple unintended scenarios may take form.

New campaign is registered by a legitimate user - Alice. The contract's owner then funds this campaign, and another legitimate user, Bob, deposits tokens. At this point, there is a data structure representing the campaign. This structure identifies Alice as the campaign's owner. It also tracks a non-zero balance for a reward token, a locked token deposited by Bob, and also a non-zero value corresponding with the lock time. A malicious user can call the `registerCampaign` function again, using the `_id` used by Alice. The validation logic will pass, because the `balance` field will be greater than 0, and the campaign will effectively be taken over by the malicious user.

Another potencial scenario may occur after a legitimate user registers a campaign. When this user is ready to deposit tokens or the admin is about to fund reward tokens, a malicious user may frontrun these transactions with a transaction of his own to reregister the campaign, effectively taking it over and eventually steal all the tokens.

Recommendation

It is recommended to change the verification mechanism to take a non-default value into consideration. An exemplary mechanism would be to use a field that will always be a non-default value for a created campaign (for instance, the `owner` field). Alternatively, another field can be introduced into the `Campaign` data structure that will reflect it's state.

Status

The issue has been fixed in `c6ba4a615f70082346fd5eda2b412f29278a4dcd`.



Possibility Of Locking Tokens In Inactive Campaigns

High

RES-BBLK-SCA02

Data Validation

Resolved

Code Section

- [contracts/MyToken.sol#L122](#)

Description

The RewardsBank contract allows users to lock their tokens in the campaigns of their choosing. In exchange for locking their tokens for a specified amount of time, users will get potentially more tokens as a reward. The `depositTokens` function is responsible for locking the user's tokens. The `depositTokens` functions contains a verification mechanism that would prevent users from locking their tokens if another user already did so before. However, this verification is faulty, as it does not prevent users from locking their tokens in inactive campaigns. The function does not verify if the campaign was actually created, as it only checks if current balance of `bbTokens` field in a corresponding structure is equal to 0. This field will be equal to 0, even if the campaign does not exist, as 0 is the default value for `uint256` type.

Additionally, if the campaign does exist, but it is already over, i.e. the `bbTokens` were already withdrawn from a given campaign, the verification mechanism present in `depositTokens` functions will pass again making it possible for users to deposit into campaigns that are already over.

Recommendation

It is recommended to introduce a new field into the data structure representing the campaign that will track it's status. The verification mechanism should then use this field and allow users to deposit their tokens only if the campaign is created, and only if it is not over yet.

Status

The issue has been fixed in `21e187a7a590a8837dec12bdf65e696a44be13f2`.



Incorrect Usage Of Nonce In Signature Replay Verification

Medium RES-BBLK-SCA03

Data Validation

Resolved

Code Section

- [contracts/MyToken.sol#L53](#)

Description

The Bonusblock's ERC721 contract implements a `setTokenURI` function that allows updating a given token's URI. The function is callable by anyone, however the arguments need to be signed and validated against a previously set public key. To prevent replay attacks, the contract utilizes nonces. However, it was observed that it does so incorrectly. The `require` statement is checking whether the nonce has already been used, and if it has, it allows the function to execute.

Recommendation

It is recommended to change the `require` statement to a following:

```
require(nonces[nonce] == false, "Nonce has been used");
```

And then update the mapping to a `true` as it is currently implemented. This will assure that a given nonce can only be used once.

Status

The issue has been fixed in [ea69f5d2992a600596e2c0d920b5efcc84dfb103](#).



Possibility of Funding Non-Existent Campaigns

Medium RES-BBLK-SCA04

Data Validation

Resolved

Code Section

- `contracts/MyToken.sol#L116`

Description

The Bonusblock's RewardsBank contract is responsible for handling the campaign logic. Each campaign requires funding. The `fundCampaign` function contains a `require` statement that was meant to assure that funding would be possible only for the existing campaigns. However, the check used for this verification checks if an `uint256` value is bigger or equal to 0, which is always true. The vulnerability originates from the fact that mappings for keys that were not previously inserted will return default values. This principle applies to `structs` as well - making it return a structure with all of its members be default values.

Recommendation

It is recommended to change the verification mechanism. An exemplary mechanism would be to use a field that will always be a non-default value for a created campaign (for instance, the `owner` field). Alternatively, another field can be introduced into the `Campaign` data structure that will reflect it's state.

Status

The issue has been fixed in 21e187a7a590a8837dec12bdf65e696a44be13f2.



Missing Public Key Validation

Low	RES-BBLK-SCA05	Data Validation	Resolved
-----	----------------	-----------------	----------

Code Section

- `contracts/MyToken.sol#L25-L27`

Description

The `MyToken` ERC721 contract defines an `updateKey` function. It is used to update the `publicKey` variable, which in turn is used to verify signatures. However, the `updateKey` function is missing the length check on the provided `newKey`.

Recommendation

It is recommended to add a `require` statement that will assure the valid length of a new public key.

Status

The issue has been fixed in 091fbdce5a2c8f9cf3f4eda9995bfc81e86dd675.



Unused Functions

Info

RES-BBLK-SCA06

Code Quality

Resolved

Code Section

- [contracts/MyToken.sol#L227-L236](#)

Description

It was observed that the RewardsBank contract defines a `calculatePercentage` function, however this function is not used anywhere in the contract.

Recommendation

It is recommended to delete unused functions to save in storage gas costs when deploying the contract.

Status

The issue has been fixed in [21e187a7a590a8837dec12bdf65e696a44be13f2](#).



Floating Pragma

Info

RES-BBLK-SCA07

Code Quality

Resolved

Code Section

Not specified

Description

Floating pragmas is a feature of Solidity that allows developers to specify a range of compiler versions that can be used to compile the smart contract code. For security purposes specifically, the usage of floating pragmas is discouraged and not recommended. Contracts should be compiled and deployed with a strict pragma, the one which was thoroughly tested the most by developers. Locking the pragma helps ensure that contracts do not accidentally get deployed using too old or too recent compiler versions that might introduce security issues that impact the contract negatively.

It should be noted however that, pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package.

The smart contracts of Bonusblock's protocol make use of floating pragmas. These are not intended for use as libraries for other developers and, as such, should be locked of their pragma.

Recommendation

It is recommended to lock the pragma of all smart contracts to the same Solidity compiler version.

Status

The issue has been fixed.



Missing Usage Of ERC165 Standard

Info

RES-BBLK-SCA08

Code Quality

Acknowledged

Code Section

Not specified

Description

It was observed that the Bonusblock contracts do not leverage ERC165 standard. Using ERC165 assures that a contract deployed to a given address implements expected functionalities. It also indirectly performs a check for zero-address. Using ERC165 is not a requirement for a contract to be implemented correctly, however using it limits the impact of a human error during contract's configuration and it is considered a good practice.

Recommendation

It is recommended to leverage the ERC165 standard wherever possible. Each time a contract handles an address that will be used for cross-contract calls, and the interface of the contract is known, the `supportsInterface()` function defined by ERC165 standard should be used.

Status

The issue was acknowledged by Bonusblock's team. The development team stated "Will include in future release."



Redundant Code On `verifySignature()`

Info

RES-BBLK-SCA09

Gas Optimization

Acknowledged

Code Section

- [contracts/MyToken.sol#L215-L224](#)
- [contracts/MyToken.sol#L41-L47](#)

Description

It was observed that both `RewardsBank` and `MyToken` ERC721 contract implement a `verifySignature` function. The logic of these functions is the same in both contracts. In consequence, both contracts contain this functionality, which makes them more expensive to deploy and operate. Although it is not a security vulnerability, it is considered a best practice for code quality and gas optimization to keep the shared functionalities in a `library` so that it is only deployed once on-chain and code can be reused.

Recommendation

It is recommended to create a library with `verifySignature` function's implementation and use it as a shared functionality among the contracts that require it.

Status

The issue was acknowledged by Bonusblock's team. The development team stated "Will include in future release."



Possible Reentrancy In safeMint()

Info

RES-BBLK-SCA10

Business Logic

Acknowledged

Code Section

- [contracts/MyToken.sol#L29-L38](#)

Description

The function `safeMint()` indirectly performs an arbitrary external call through `ERC721's _safeMint()`, and does not follow the Checks-Effects-Interactions pattern nor does it implement verification mechanisms against reentrancy, such as OpenZeppelin's `ReentrancyGuard`.

While it does not present an immediate security threat as it is, when further functionality is introduced, possible reentrancy scenarios may occur that may ultimately lead to financial loss on the protocol.

Recommendation

It is recommended to follow the Checks-Effects-Interactions coding pattern for all functions that inherently perform arbitrary external calls, while also implementing reentrancy verification mechanisms.

Status

The issue was acknowledged by Bonusblock's team. The development team stated "Will include in future release."



Unclaimable Ether Sent To Contract

Info

RES-BBLK-SCA11

Business Logic

Resolved

Code Section

- `custom-erc20-launch.sol#L34`

Description

The constructor of the contract possesses the keyword `payable`, which means Ether can be sent to the contract during its deployment. However, any sent Ether may not be claimed as there is no implemented functionality that allows it.

It should be noted that the severity of this finding has been reduced due to the fact that this may only happen during the deployment of the smart contract and by mistake.

Recommendation

It is recommended to implement functionality to reclaim Ether sent to contracts, either via transfers or self destruction. Otherwise, if unnecessary, `payable` keywords should be removed.

Status

The issue has been fixed in `4f95c368976d459591fe6308caccc1221a84ccf9`.

Proof of Concepts

No Proof-of-Concept was deemed relevant to describe findings in this engagement.